Fast Computation of Abelian Runs

Gabriele Fici[a,1], Tomasz Kociumaka[b,2], Thierry Lecroq[c], Arnaud Lefebvre[c], Élise Prieur-Gaston[c]

[a] *Dipartimento di Matematica e Informatica, Università di Palermo, Italy*
[b] *Faculty of Mathematics, Informatics and Mechanics, University of Warsaw, Poland*
[c] *Normandie Université, LITIS EA4108, NormaStic CNRS FR 3638, IRIB, Université de Rouen, 76821 Mont-Saint-Aignan Cedex, France*

Abstract

Given a word w and a Parikh vector $\mathcal{P}$, an abelian run of period $\mathcal{P}$ in w is a maximal occurrence of a substring of w having abelian period $\mathcal{P}$. Our main result is an online algorithm that, given a word w of length n over an alphabet of cardinality σ and a Parikh vector $\mathcal{P}$, returns all the abelian runs of period $\mathcal{P}$ in w in time $O(n)$ and space $O(\sigma + p)$, where p is the norm of $\mathcal{P}$, i.e., the sum of its components. We also present an online algorithm that computes all the abelian runs with periods of norm p in w in time $O(np)$, for any given norm p. Finally, we give an $O(n^2)$-time offline randomized algorithm for computing all the abelian runs of w. Its deterministic counterpart runs in $O(n^2 \log \sigma)$ time.

Keywords: Combinatorics on Words, Text Algorithms, Abelian Period, Abelian Run

1. Introduction

Computing maximal (non-extendable) repetitions in a word is a classical topic in the area of string algorithms (see for example [1] and references therein). Maximal repetitions of substrings, also called *runs*, give information on the repetitive regions of a word, and are used in many applications, for example in the analysis of genomic sequences.

Kolpakov and Kucherov [2] gave the first linear-time algorithm for computing all the runs in a word and conjectured that any word of length n contains less than n runs. Recently, Bannai et al. [3, 4], using the notion of Lyndon roots of a run, proved this conjecture and designed a much simpler algorithm computing the runs.

Here we deal with a generalization of this problem to the commutative setting. Recall that an abelian power is a concatenation of two or more words that have the same Parikh vector, i.e., that have the same number of occurrences of each letter of the alphabet. For example, *aababa* is an abelian square, since *aab* and *aba* both have two a's and one b, i.e., the same Parikh vector $\mathcal{P} = (2, 1)$. When an abelian power occurs within a word, one can search for its "maximal" occurrence by extending it to the left and to the right character by character without violating the condition on the number of occurrences of each letter. Following the approach of Constantinescu and Ilie [5], we say that a Parikh vector $\mathcal{P}$ is an abelian period of a word w if w can be written as $w = u_0 u_1 \cdots u_{k-1} u_k$ for some $k \geq 1$ where for $0 < i < k$ all the u_i's have the same Parikh vector $\mathcal{P}$ and the Parikh vectors of u_0 and u_k are contained in $\mathcal{P}$. If $k > 2$, we say that the

Email addresses: Gabriele.Fici@unipa.it (Gabriele Fici), kociumaka@mimuw.edu.pl (Tomasz Kociumaka), Thierry.Lecroq@univ-rouen.fr (Thierry Lecroq), Arnaud.Lefebvre@univ-rouen.fr (Arnaud Lefebvre), Elise.Prieur@univ-rouen.fr (Élise Prieur-Gaston)

[1] Supported by the Italian Ministry of Education (MIUR) project PRIN 2010/2011 "Automi e Linguaggi Formali: Aspetti Matematici e Applicativi".

[2] Supported by the Polish Ministry of Science and Higher Education under the 'Iuventus Plus' program in 2015-2016 grant no 0392/IP3/2015/73. The author is also supported by Polish budget funds for science in 2013-2017 as a research project under the 'Diamond Grant' program.

word w is periodic with period $\mathcal{P}$. Note that the factorization above is not necessarily unique. For example, $a \cdot bba \cdot bba \cdot \varepsilon$ and $\varepsilon \cdot abb \cdot abb \cdot a$ (ε denotes the empty word) are two factorizations of the word $abbabba$ both corresponding to the abelian period $(1, 2)$. Moreover, the same word can have different abelian periods.

In this paper we define an *abelian run* of period $\mathcal{P}$ in a word w as an occurrence of a substring v of w such that v is periodic with abelian period $\mathcal{P}$ and this occurrence cannot be extended to the left nor to the right by one letter into a substring periodic with period $\mathcal{P}$.

For example, let $w = ababaaa$. Then the prefix $ab \cdot ab \cdot a = w[0..4]$ has abelian period $(1, 1)$ but it is not an abelian run since the prefix $a \cdot ba \cdot ba \cdot a = w[0..5]$ also has abelian period $(1, 1)$. The latter, on the other hand, is an abelian run of period $(1, 1)$ in w.

Looking for abelian runs in a word can be useful to detect regions in the word where there is some kind of non-exact repetitiveness, for example regions with several consecutive occurrences of a substring or its reversal.

Matsuda et al. [6] recently presented an offline algorithm for computing all abelian runs of a word of length n in $O(n^2)$ time. Notice that, however, the definition of abelian run in [6] is slightly different from the one we consider here. We compare both versions in Section 2. Basically, our notion of abelian run is more restrictive than the one of [6], for which we use the term "anchored run".

We first present an online algorithm that, given a word w of length n over an alphabet of cardinality σ and a Parikh vector $\mathcal{P}$, returns all the abelian runs of period $\mathcal{P}$ in w in time $O(n)$ and space $O(\sigma + p)$, where p is the norm of $\mathcal{P}$, that is, the sum of its components. This algorithm improves upon the one given in [7] which runs in time $O(np)$. Next, we give an $O(np)$-time online algorithm for computing all the abelian runs with periods of norm p of a word of length n, for any given p. Finally, we present an $O(n^2)$ (resp. $O(n^2 \log n)$) -time offline randomized (resp. deterministic) algorithm for computing all the abelian runs of a word of length n.

The rest of this article is organized as follows. Sect. 2 introduces central concepts and fixes the notation. In Sect. 3 we review the results on abelian runs given in [6]. Sect. 4 is devoted to the presentation of our main result: a new solution for computing the abelian runs for a given Parikh vector. In Sect. 5 we apply this algorithm in a procedure for computing the abelian runs with periods of a given norm. Next, in Sect. 6, we design a solution for computing all the abelian runs, which builds upon the result recalled in Sect. 3. Finally, we conclude in Sect. 7.

2. Definitions and Notation

Let $\Sigma = \{a_1, a_2, \ldots, a_\sigma\}$ be a finite ordered alphabet of cardinality σ, and let Σ^* be the set of finite words over Σ. We assume that the mapping between a_i and i can be evaluated in constant time for $1 \leq i \leq \sigma$. We let $|w|$ denote the length of the word w. Given a word $w = w[0..n-1]$ of length $n > 0$, we write $w[i]$ for the $(i+1)$-th symbol of w and, for $0 \leq i \leq j < n$, we write $w[i..j]$ to denote a fragment of w from the $(i+1)$-th symbol to the $(j+1)$-th symbol, both included. This fragment is an occurrence of a substring $w[i] \cdots w[j]$. For $0 \leq i \leq n$, $w[i..i-1]$ denotes the empty fragment. We let $|w|_a$ denote the number of occurrences of the symbol $a \in \Sigma$ in the word w. The Parikh vector of w, denoted by $\mathcal{P}_w$, counts the occurrences of each letter of Σ in w, that is, $\mathcal{P}_w = (|w|_{a_1}, \ldots, |w|_{a_\sigma})$. Notice that two words have the same Parikh vector if and only if one word is a permutation (i.e., an anagram) of the other. Given the Parikh vector $\mathcal{P}_w$ of a word w, we let $\mathcal{P}_w[i]$ denote its i-th component and $|\mathcal{P}_w|$ its norm, defined as the sum of its components. Thus, for $w \in \Sigma^*$ and $1 \leq i \leq \sigma$, we have $\mathcal{P}_w[i] = |w|_{a_i}$ and $|\mathcal{P}_w| = \sum_{i=1}^{\sigma} \mathcal{P}_w[i] = |w|$. Finally, given two Parikh vectors $\mathcal{P}, \mathcal{Q}$, we write $\mathcal{P} \subseteq \mathcal{Q}$ if $\mathcal{P}[i] \leq \mathcal{Q}[i]$ for every $1 \leq i \leq \sigma$. If additionally $\mathcal{P} \neq \mathcal{Q}$, we write $\mathcal{P} \subset \mathcal{Q}$ and say that $\mathcal{P}$ is contained in $\mathcal{Q}$.

Definition 1 (Abelian period [5]). A factorization $w = u_0 u_1 \cdots u_{k-1} u_k$ satisfying $k \geq 1$, $\mathcal{P}_{u_1} = \cdots = \mathcal{P}_{u_{k-1}} = \mathcal{P}$, and $\mathcal{P}_{u_0} \subset \mathcal{P} \supset \mathcal{P}_{u_k}$ is called a *periodic factorization* of w with respect to $\mathcal{P}$. If a word w admits such a factorization, we say that $\mathcal{P}$ is an *(abelian) period* of w.

We call fragments u_0 and u_k respectively the *head* and the *tail* of the factorization, while the remaining factors are called *cores*. Note that the head and the tail are of length strictly smaller than $|\mathcal{P}|$; in particular they can be empty.

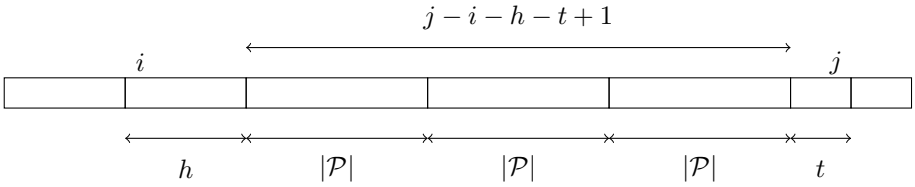

Figure 1: The tuple (i, h, t, j) denotes an occurrence of a substring starting at position i, ending at position j, and having abelian period $\mathcal{P}$ with head length h and tail length t.

Observe that a periodic factorization with respect to a fixed period is not unique. However, it suffices to specify $|u_0|$ to indicate a particular factorization; see [5]. Dealing with factorizations of fragments of a fixed text, it is more convenient to use a different quantity for this aim. Suppose $w[i..j] = u_0 \cdots u_k$ is a factorization with respect to abelian period $\mathcal{P}$ with $p = |\mathcal{P}|$. Observe that consecutive starting positions $i_1, \ldots, i_k$ of factors $u_1, \ldots, u_k$ differ by exactly p. Hence, they share a common remainder modulo p, which we call the *anchor* of the factorization. Note that the anchor does not change if we trim a factorization of $w[i..j]$ to a factorization of a shorter fragment, or if we extend it to a factorization of a longer fragment.

Definition 2 (Anchored period). A fragment $w[i..j]$ has an abelian period $\mathcal{P}$ *anchored* at k if it has a periodic factorization with respect to $\mathcal{P}$ whose anchor is $k \bmod p$.

If w has a factorization with at least two cores, we say that w is *periodic* with period $\mathcal{P}$ (anchored at k if $k \bmod p$ is the anchor of the factorization).

Definition 3 (Abelian run). A fragment $w[i..j]$ is called an *abelian run* with period $\mathcal{P}$ if it is periodic with period $\mathcal{P}$ and maximal with respect to this property (i.e., each of $w[i-1..j]$ and $w[i..j+1]$ either does not exist or it is not periodic with period $\mathcal{P}$).

We shall often represent an abelian run $w[i..j]$ as a tuple (i, h, t, j) where h and t are respectively the lengths of the head and the of the tail of a periodic factorization of $w[i..j]$ with period $\mathcal{P}$ and at least two cores (see Figure 1). Note that $(i + h) \bmod p$ is the anchor of the factorization, and that $\frac{1}{p}(j - i - h - t - 1)$ is the number of cores, in particular it is an integer.

Observe that an abelian run with period $\mathcal{P}$ may have several valid factorizations. For example, $a \cdot ba \cdot ba \cdot \varepsilon$ and $\varepsilon \cdot ab \cdot ab \cdot a$ are factorization of a run $w[0..4]$ with period $\mathcal{P} = (1, 1)$ in $w = ababa$. Therefore the run can be represented as $(0, 1, 0, 4)$ and as $(0, 0, 1, 4)$. However, in $v = abab$ only $(0, 0, 0, 3)$ is a representation of $v[0..3]$ as an abelian run with period $\mathcal{P} = (1, 1)$. This is because $(0, 1, 1, 3)$ corresponds to a factorization $v[0..3] = a \cdot ba \cdot b$ with one core only, and such a factorization does not indicate that $v[0..3]$ is an abelian run.

Matsuda et al. [6] gave a different definition of abelian runs, where maximality is with respect to extending a fixed factorization. In this paper, we call such fragments anchored (abelian) runs.

Definition 4 (Anchored run [6]). A fragment $w[i..j]$ is a *k-anchored abelian run* with period $\mathcal{P}$ if $w[i..j]$ is periodic with period $\mathcal{P}$ anchored at k and maximal with respect to this property (i.e., each of $w[i-1..j]$ and $w[i..j+1]$ either does not exist or it is not periodic with period $\mathcal{P}$ anchored at k).

Note that every abelian run is an anchored run with the same period (for some anchor). The converse is not true, since it might be possible to extend an anchored run preserving the period but not the anchor. For example, in the word $w = ababaaa$ considered in the introduction, the fragment $w[0..4] = \varepsilon \cdot ab \cdot ab \cdot a$ is a 0-anchored run but not an abelian run, since $w[0..5] = a \cdot ba \cdot ba \cdot a$ is periodic with abelian period $(1, 1)$.

Since a factorization is uniquely determined by the anchor, standard inclusion-maximality is equivalent to the condition in Definition 4.

Observation 5. *Let $w[i..j]$ and $w[i'..j']$ be fragments of w with abelian period $\mathcal{P}$ anchored at k. If $w[i..j]$ is properly contained in $w[i'..j']$ (i.e, $i' < i$ and $j \leq j'$, or $i' \leq i$ and $j < j'$), then $w[i..j]$ is not a k-anchored abelian run with period $\mathcal{P}$.*

Abelian runs enjoy the same property, but its proof is no longer trivial.

Lemma 6. *Let $w[i..j]$ and $w[i'..j']$ be fragments of w with abelian period $\mathcal{P}$. If $w[i..j]$ is properly contained in $w[i'..j']$, then $w[i..j]$ is not an abelian run with period $\mathcal{P}$.*

Proof. We assume that $i' < i$. The case of $j < j'$ is symmetric. For a proof by contradiction suppose that $w[i..j]$ is an abelian run and let $w[i..j] = u_0 \cdots u_k$ be a periodic factorization with period $\mathcal{P}$ and at least two cores (i.e., satisfying $k \geq 3$). A periodic factorization of $w[i'..j']$ can be trimmed to a factorization $w[i-1..j] = v_0 \cdots v_\ell$. However, since $w[i..j]$ is an abelian run, this factorization must have at most one core (i.e., $\ell \leq 2$). Moreover, $u_0 \cdots u_k$ cannot be extended to a factorization of $w[i-1..j] = u'_0 u_1 \cdots u_k$. In other words u'_0, the extension of u_0 by one letter to the left, must satisfy $\mathcal{P}_{u'_0} \not\subseteq \mathcal{P}$.

Let $p = |\mathcal{P}|$. The conditions on the number of cores imply $|u_0|+2p \leq |w[i..j]|$ and $|w[i-1]..j]| < |v_0|+2p$. Consequently, $|u'_0| = |u_0| + 1 < |v_0|$, i.e., u'_0 is a proper prefix of v_0. This yields $\mathcal{P}_{u'_0} \subset \mathcal{P}_{v_0} \subset \mathcal{P}$, which is in contradiction with $\mathcal{P}_{u'_0} \not\subseteq \mathcal{P}$. □

Corollary 7. *Let w be a word. For a fixed Parikh vector $\mathcal{P}$, there is at most one abelian run with abelian period $\mathcal{P}$ starting at each position of w.*

3. Previous Work

Matsuda et al. [6] presented an algorithm that computes all the anchored runs of a word w of length n in $O(n^2)$ time and space complexity. The initial step of the algorithm is to compute maximal abelian powers in w. Recall that an abelian power is a concatenation of several abelian-equivalent words. In other words, an abelian power of period $\mathcal{P}$ is a word admitting a periodic factorization with respect to $\mathcal{P}$ with an empty head, an empty tail and at least two cores. A fragment $w[i..j]$ is a maximal abelian power if it cannot be extended to a longer power of period $\mathcal{P}$ (preserving the anchor). Formally, the maximality conditions are

1. $\mathcal{P}_{w[i-p..i-1]} \neq \mathcal{P}_{w[i..i+p-1]}$ or $i - p < 0$, and
2. $\mathcal{P}_{w[j-p+1..j]} \neq \mathcal{P}_{w[j+1..j+p]}$ or $j + p \geq n$,

where $p = |\mathcal{P}|$.

The approach of [6] is to first compute all the abelian squares using the algorithm by Cummings & Smyth [8]. The next step is to group squares into maximal abelian powers. For this, it suffices to merge pairs of overlapping abelian squares of the form $w[i..i+2p-1]$ and $w[i+p..i+3p-1]$. This way maximal abelian powers are computed in $O(n^2)$ time.

Observe that there is a natural one-to-one correspondence between maximal abelian powers and anchored runs: it suffices to trim the head and the tail of the factorization of an anchored run to obtain a maximal abelian power. Hence, the last step of the algorithm is to compute the maximal head and tail by which each abelian power can be extended. This could be done naively in $O(n^3)$ time overall, but a clever computation enables to find all the abelian runs in time and space $O(n^2)$ (see [6] for further details).

In Section 6, we extend this result to compute the abelian runs only rather than all the anchored runs. Both these algorithms work offline: they need to know the whole word before reporting any abelian run. In the following two sections we give several online algorithms, which are able to report a run ending at position $i-1$ of a word w before reading $w[i+1]$ and the following letters. Clearly, not knowing $w[i]$ one cannot decide whether the run could be extended to the right, so this is the optimal delay. However, these methods are restricted to finding runs of a given period or a given norm of the periods, respectively.

4. Computing Abelian Runs with Fixed Parikh Vector

In this section we present our online solution for computing all the abelian runs of a given Parikh vector $\mathcal{P}$ of norm p in a given word w. The algorithm works in $O(n)$ time and $O(\sigma + p)$ space where $n = |w|$.

First, in Sect. 4.1, we show how to compute all anchored runs of period $\mathcal{P}$. Later, in Sect. 4.2, we modify the algorithm to return abelian runs only. We conclude in Sect. 4.3 with an example course of actions in our solution.

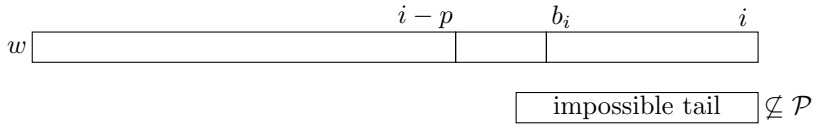

Figure 2: $B_i[k] = \infty$ for $i - p + 1 < k < b_i$.

4.1. Algorithm for Anchored Runs

We begin with a description of data maintained while scanning the string w. For an integer k, let $B_i[k]$ be the starting position of the longest suffix of $w[0..i]$ which has period $\mathcal{P}$ anchored at k. If there is no such a suffix, we set $B_i[k] = \infty$. Since this notion depends on k mod p only, we store $B_i[k]$ for $0 \le k < p$ only.

Let b_i be the starting position of the longest suffix of $w[0..i]$ whose Parikh vector is contained in or equal to $\mathcal{P}$. In other words, we have $\mathcal{P}_{w[b_i..i]} \subseteq \mathcal{P}$ and $\mathcal{P}_{w[b_i-1..i]} \not\subseteq \mathcal{P}$ (or $b_i = 0$). Note that $b_i = i + 1$ if $w[i]$ does not occur in $\mathcal{P}$.

Observe that the tail of any periodic factorization of a suffix of $w[0..i]$ must be contained in $w[b_i..i]$. This leads to the following characterization:

Lemma 8. *Let $0 \le i < |w|$. We have $B_i[k] \le k$ for $b_i \le k \le i + 1$ and $B_i[k] = \infty$ for $i - p + 1 < k < b_i$.*

Proof. For $b_i \le k \le i + 1$, the fragment $w[k..i]$ has abelian period $\mathcal{P}$ anchored at k. (The underlying factorization has empty head, no cores and tail $w[k..i]$, unless $k = b_i = i - p + 1$, when the factorization has one core, empty head and empty tail). Hence, we have $B_i[k] \le k$ directly from the definition.

For $i - p + 1 < k < b_i$, the tail of the factorization with anchor k mod p would need to start at position k, which is impossible (see Figure 2). □

The values b_{i-1} and b_i are actually sufficient to describe B_i based on B_{i-1}.

Lemma 9. *For $0 \le i < |w|$ the following equalities hold:*

1. $B_i[k] = \infty \ne B_{i-1}[k]$ *for* $\max(i - p + 1, b_{i-1}) \le k < b_i$,
2. $B_i[k] = B_{i-1}[k]$ *for $b_i \le k \le i$ and for $i - p + 1 < k < b_{i-1}$,*
3. $B_i[i + 1] = b_i$ *if $b_i > i - p + 1$ and $B_i[i + 1] = B_{i-1}[i - p + 1]$ otherwise.*

Proof. Lemma 8 implies that $B_i[k] = \infty$ for $i - p + 1 < k < b_i$ and $B_{i-1}[k] = \infty$ for $i - p + 1 < k < b_{i-1}$ (hence $B_{i-1}[k] = B_i[k]$ in this latter case). For $b_i \le k < i$, we have $\mathcal{P}_{w[k..i]} \subseteq \mathcal{P}$, so we can extend the factorization of a suffix of $w[0..i - 1]$ whose tail starts at position k (see Figure 3).

Finally, note that $B_i[i + 1]$ is the starting position of the maximal suffix of $w[0..i]$ with an empty-tail periodic factorization. If $\mathcal{P}_{w[i-p+1..i]} \ne \mathcal{P}$ (i.e., if $b_i > i - p + 1$), this is just $w[b_i..i]$. Otherwise, we can extend the factorization of a suffix of $w[0..i - 1]$ whose tail starts at position $i - p + 1$. □

Having read letter $w[i]$, we need to report anchored runs which end at position $i - 1$. For this, we use the following characterization.

Lemma 10. *Let $i - p < k \le i$. A fragment $w[b..i - 1]$ is a k-anchored run with period $\mathcal{P}$ if and only if $B_{i-1}[k] = b \le k - 2p$ and $B_i[k] > b$.*

Proof. Clearly an anchored run ending at position $i - 1$ must be a left-maximal suffix of $w[0..i - 1]$ with a given anchor. Moreover, we must have $b \le k - 2p$ so that the factorization has at least two cores and $B_i[k] > b$ due to right-maximality. It is easy to see that these conditions are sufficient. □

By Lemma 9, most entries of B_i are inherited from B_{i-1}, so we use a single array B and having read $w[i]$, we update its entries. As evident from Lemma 10, each anchored run to be reported corresponds to a modified entry.

The algorithm AnchoredRun$(\mathcal{P}, p, w, n)$ in Figure 4 implements our approach. The **while** loop increments k from b_{i-1} to b_i. For $k > i - p$, we set $B[k]$ to ∞ and possibly report a run. Note that $k = i - p + 1$ is within the scope of Case 3 rather than Case 1 in Lemma 9. However, later we set $B[i + 1]$ to b_i if $b_i > i - p + 1$ (as described in Case 3). Nevertheless, if an $(i + 1)$-anchored run needs to be reported, we have $B_{i-1}[i - p + 1] < \infty = B_i[i - p + 1]$, so $b_{i-1} \leq i - p + 1$ and thus $k = i - p + 1$ is considered in the loop.

Theorem 11. *The algorithm* AnchoredRun$(\mathcal{P}, p, w, n)$ *computes all the anchored runs with period* $\mathcal{P}$ *of norm* p *in a word* w *of length* n *in time* $O(n)$ *and additional space* $O(\sigma + p)$.

Proof. The correctness of the algorithm comes from Lemmas 9-10 and the discussion above. The external **for** loop in lines 3–14 runs $n+1$ times. The internal **while** loop in lines 4–12 cannot iterate more than $n+1$ times since it starts with k equal to 0 and ends when k is equal to n and k can only be incremented by 1 (in line 12). The test $\mathcal{P}_{w[k..i]} \not\subseteq \mathcal{P}$ in line 4 can be realized in constant time once we store $\mathcal{P}_{w[k..i]}$ and a counter of its components for which the value is greater than in $\mathcal{P}$. This data needs to be updated once we increment i in the **for** loop and k in line 12. We then need to increment the component $w[i]$ or decrement the component $w[k]$ of $\mathcal{P}_{w[k..i]}$, respectively. The global counter needs to be updated accordingly. All the

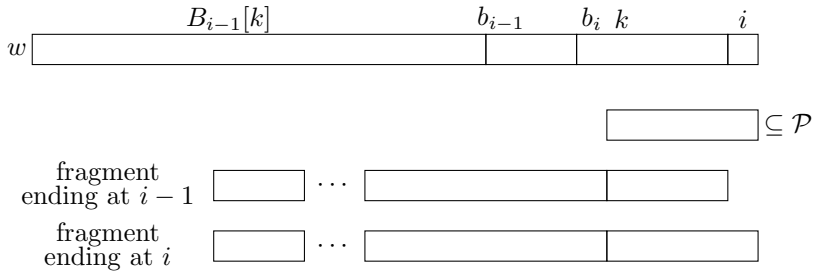

Figure 3: $B_i[k] = B_{i-1}[k]$ for $b_i \leq k \leq i$.

```
AnchoredRun(P, p, w, n)
 1  k ← 0
 2  B[0] ← k
 3  for i ← 0 to n do
 4  |  while k ≤ n and (i = n or P_{w[k..i]} ⊈ P) do
 5  |  |  if k > i − p then
 6  |  |  |  b ← B[k mod p]
 7  |  |  |  B[k mod p] ← ∞
 8  |  |  |  if b ≤ k − 2p then
 9  |  |  |  |  h ← (k − b) mod p
10  |  |  |  |  t ← i − k
11  |  |  |  |  Output(b, h, t, i − 1)
12  |  |  k ← k + 1
13  |  if k > i − p + 1 then
14  |  |  B[(i + 1) mod p] ← k
```

Figure 4: Algorithm computing all the anchored runs of period $\mathcal{P}$ of norm p in a word w of length n.

other operations run in constant time. Thus the total time complexity of the algorithm is $O(n)$. The space complexity comes from the number of counters (σ) and the size of the array B (p). □

Note that the space consumption can be reduced to $O(p)$ at the price of introducing (Monte Carlo) randomization. Instead of storing the Parikh vectors in a plain form, we can use dynamic hash tables [9] so that the size is proportional to the number of non-zero entries.

4.2. Algorithm for Abelian Runs

In this section we extend our algorithm so that it reports abelian runs only. For an offline solution, we could simply determine the anchored runs (using the procedure developed above) and filter out those which are not maximal. However, in order to obtain an online algorithm, we need a more subtle approach, which is based on the following characterization.

Lemma 12. *A fragment $w[b..i-1]$ is an abelian run with period $\mathcal{P}$ if and only if it is an anchored run (with period $\mathcal{P}$) and for each k' the inequalities $B_{i-1}[k'] \geq b$ and $B_i[k'] > b$ hold.*

Proof. By Lemma 6, an abelian run of period $\mathcal{P}$ cannot be properly contained in a fragment with period $\mathcal{P}$ (anchored at some k'). Conditions involving $B_{i-1}[k']$ and $B_i[k']$ enforce left-maximality and right-maximality, respectively. Since each abelian run is an anchored run (with the same period) and since all anchored runs are periodic, the claim follows. □

To apply Lemma 12, it suffices to find an anchor k such that $b = B_{i-1}[k] = \min_{k'} B_{i-1}[k'] < \min_{k'} B_i[k']$. There can be several such anchors and in case of ties we are going to detect the one for which the factorization of $w[b..i-1]$ has shortest tail. This factorization maximizes the number of cores, so if $w[b..i-1]$ is an anchored run with any anchor, it is with that one in particular. Note that the **while** loop in lines 4–12 of Algorithm AnchoredRun processes anchors $b_{i-1} \leq k < b_i$ in the order of decreasing tail lengths and

```
Run(P, p, w, n)
 1  k ← 0
 2  L ← ∅
 3  B[0] ← k
 4  Ptr[0] ← insertAtTheEnd(L, 0)
 5  for i ← 0 to n do
 6  |  b_min ← B[getFirst(L)]
 7  |  while k ≤ n and (i = n or P_{w[k..i]} ⊈ P) do
 8  |  |  if k > i − p then
 9  |  |  |  b ← B[k mod p]
10  |  |  |  B[k mod p] ← ∞
11  |  |  |  Delete(L, Ptr[k mod p])
12  |  |  |  Ptr[k mod p] ← Nil
13  |  |  |  if b = b_min and B[getFirst(L)] > b and b ≤ k − 2p then
14  |  |  |  |  h ← (k − b) mod p
15  |  |  |  |  t ← i − k
16  |  |  |  |  Output(b, h, t, i)
17  |  |  k ← k + 1
18  |  |  if k > i − p + 1 then
19  |  |  |  B[(i + 1) mod p] ← k
20  |  |  |  Ptr[(i + 1) mod p] ← insertAtTheEnd(L, (i + 1) mod p)
```

Figure 5: Algorithm computing all the abelian runs of period $\mathcal{P}$ of norm p in a word w of length n.

updates the underlying values $B[k]$ from $B_{i-1}[k]$ to $B_i[k]$. For the sought anchor k this update strictly increases the value $\min_{k'} B[k']$, and moreover this is the first increase of the minimum within a given iteration of the outer **for** loop. Hence, we record the original minimum and check for an abelian run only if $\min_{k'} B[k']$ increases from that value.

To implement the procedure described above, we need to efficiently compute the smallest element in the array B. For this, recall that $B[j]$ can only be modified from ∞ to k (and the value k does not decrease throughout the algorithm) or from some value back to ∞. We maintain a doubly-linked list L of all indices j with finite $B[j]$ such that the order of indices j in the list is consistent with the order of values $B[j]$. To update the list, it suffices to insert the index j to the end of list while setting $B[j]$ to k, and remove it from the list setting $B[j]$ to ∞. Then the smallest value in B is attained at an argument stored as the first element of the list L (or ∞, if the list is empty).

The algorithm $\textsc{Run}(\mathcal{P}, p, w, n)$, depicted in Figure 5, implements the approach described above. It uses the following constant-time functions to operate on lists:

- $\textsc{insertAtTheEnd}(L, e)$ that inserts e at the end of the doubly-linked list L and returns a pointer to the location of e in the list;
- $\textsc{Delete}(L, ptr)$ that deletes the element pointed by ptr from the doubly-linked list L;
- $\textsc{getFirst}(L)$ that returns the first element of the list L (0 if the list is empty).

The algorithm also uses an array Ptr which maps any anchor j to a pointer of the corresponding location in the list L (or *Nil* if $B[j] = \infty$).

The discussion above proves that $\textsc{Run}(\mathcal{P}, p, w, n)$ correctly computes abelian runs with period $\mathcal{P}$ in w. Its running time is the same as that of $\textsc{AnchoredRun}(\mathcal{P}, p, w, n)$ since the structure of the computations remains the same while additional instructions run in constant time. Memory consumption is still $O(p + \sigma)$ because both L and Ptr take $O(p)$ space.

Theorem 13. *The algorithm* $\textsc{Run}(\mathcal{P}, p, w, n)$ *computes all the abelian runs with period* $\mathcal{P}$ *of norm* p *in a word* w *of length* n *in time* $O(n)$ *and additional space* $O(\sigma + p)$*, which can be reduced to* $O(p)$ *using randomization.*

4.3. Example

Let us see the behaviour of the algorithm on $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $w = \mathtt{abaababaabbb}$ and $\mathcal{P} = (2, 2)$:

$k = 0 \quad B = [0, \infty, \infty, \infty] \quad L = (0)$
$i = 0 \quad \mathcal{P}_{w[0]} \subseteq \mathcal{P} \quad B = [0, 0, \infty, \infty] \quad L = (0, 1)$
$i = 1 \quad \mathcal{P}_{w[0..1]} \subseteq \mathcal{P} \quad B = [0, 0, 0, \infty] \quad L = (0, 1, 2)$
$i = 2 \quad \mathcal{P}_{w[0..2]} \subseteq \mathcal{P} \quad B = [0, 0, 0, 0] \quad L = (0, 1, 2, 3)$
$i = 3 \quad \mathcal{P}_{w[0..3]} \not\subseteq \mathcal{P} \quad b = 0 \quad B = [\infty, 0, 0, 0] \quad L = (1, 2, 3) \quad k = 1$
$\mathcal{P}_{w[1..3]} \subseteq \mathcal{P} \quad B = [1, 0, 0, 0] \quad L = (1, 2, 3, 0)$
$i = 4 \quad \mathcal{P}_{w[1..4]} \subseteq \mathcal{P}$
$i = 5 \quad \mathcal{P}_{w[1..5]} \not\subseteq \mathcal{P} \quad k = 2$
$\mathcal{P}_{w[2..5]} \not\subseteq \mathcal{P} \quad b = 0 \quad B = [1, 0, \infty, 0] \quad L = (1, 3, 0) \quad k = 3$
$\mathcal{P}_{w[3..5]} \subseteq \mathcal{P} \quad B = [1, 0, 3, 0] \quad L = (1, 3, 0, 2)$
$i = 6 \quad \mathcal{P}_{w[3..6]} \subseteq \mathcal{P}$
$i = 7 \quad \mathcal{P}_{w[3..7]} \not\subseteq \mathcal{P} \quad k = 4$
$\mathcal{P}_{w[4..7]} \subseteq \mathcal{P}$
$i = 8 \quad \mathcal{P}_{w[4..8]} \not\subseteq \mathcal{P} \quad k = 5$
$\mathcal{P}_{w[5..8]} \not\subseteq \mathcal{P} \quad b = 0 \quad B = [1, \infty, 3, 0] \quad L = (3, 0, 2) \quad k = 6$
$\mathcal{P}_{w[6..8]} \subseteq \mathcal{P} \quad B = [1, 6, 3, 0] \quad L = (3, 0, 2, 1)$
$i = 9 \quad \mathcal{P}_{w[6..9]} \subseteq \mathcal{P}$
$i = 10 \quad \mathcal{P}_{w[6..10]} \not\subseteq \mathcal{P} \quad k = 7$

$\mathcal{P}_{w[7..10]} \subseteq \mathcal{P}$

$i = 11$ $\mathcal{P}_{w[7..11]} \not\subseteq \mathcal{P}$ $k = 8$

$\mathcal{P}_{w[8..11]} \not\subseteq \mathcal{P}$ $b = 1$ $B = [\infty, 6, 3, 0]$ $L = (3, 2, 1)$ $k = 9$

$\mathcal{P}_{w[9..11]} \not\subseteq \mathcal{P}$ $b = 6$ $B = [\infty, \infty, 3, 0]$ $L = (3, 2)$ $k = 10$

$\mathcal{P}_{w[10..11]} \subseteq \mathcal{P}$ $B = [10, \infty, 3, 0]$ $L = (3, 2, 0)$

$i = 12$

$b = 3$ $B = [10, \infty, \infty, 0]$ $L = (3, 0)$ $k = 11$

$b = 0$ $B = [10, \infty, \infty, \infty]$ $L = (0)$

$h = 3$ $t = 1$ $\textsc{Output}(0, 3, 1, 11)$ $k = 12$

$b = 10$ $B = [\infty, \infty, \infty, \infty]$ $L = ()$ $k = 13$

5. Computing Abelian Runs with Fixed Parikh Vector Norm

In this section we develop an $O(np)$-time algorithm to compute all abelian runs with periods of norm p. First, we describe the algorithm for anchored runs and later generalize it to abelian runs.

5.1. Anchored Runs

Let us start with a simple offline algorithm which works in $O(n)$ time to compute k-anchored runs with period of norm p for fixed values p and k. This method is similar to the algorithm of Matsuda et al. [6] briefly described in Section 3. Namely, it suffices to compute maximal abelian powers with periods of norm p anchored at k, and then extend them by a head and a tail.

Define a *block* as any fragment of the form $w[i..i+p-1]$ such that $i \equiv k \pmod{p}$. Note that the cores in decompositions with anchor $k \bmod p$ are blocks. Finding k-anchored powers with periods of a given norm p is very easy if the anchor is fixed. We consider consecutive blocks, naively check if they are abelian-equivalent and merge any maximal chains of abelian-equivalent blocks. Determining the head and the tail of the k-anchored runs is also simple. For each $i \equiv k \pmod{p}$ we compute the longest suffix of $w[0..i-1]$ and the longest prefix of $w[i+p..n-1]$ whose Parikh vectors are contained in $\mathcal{P}_{w[i..i+p-1]}$.

This approach can be implemented online in $O(\sigma + p)$ space as follows: we scan consecutive blocks and (naively) check their abelian equivalence. Whenever we read a full block (say, starting at position i), we compute the longest suffix $w[b_{i-1}..i-1]$ of $w[0..i-1]$ whose Parikh vector is contained in $\mathcal{P}_{w[i..i+p-1]}$. This gives a periodic factorization of $w[b_{i-1}..i+p-1]$ anchored at $k \bmod p$. We then try to extend it to the right while reading further characters. Once it is impossible to extend the factorization, say by letter $w[j+1]$, we declare $w[b_{i-1}..j]$ as a maximal fragment with period $\mathcal{P}_{w[i..i+p-1]}$ anchored at k. If the decomposition has at least two cores, we report an anchored run. If we succeed to extend by a full block (i.e., if $\mathcal{P}_{w[i..i+p-1]} = \mathcal{P}_{w[i+p..i+2p-1]}$), we do not restart the algorithm but instead we continue to extend the factorization. This way, we guarantee that $b_{i-1} > i - p$ whenever we start building a new factorization.

Clearly, the procedure described above computes all k-anchored runs with period of norm p. To compute all anchored runs, we simply run it in parallel for all p possible anchors.

Theorem 14. *There is an algorithm which computes online all the anchored runs with periods of norm p in a word w of length n over an alphabet of size σ in time $O(np)$ and additional space $O(p(\sigma + p))$, which can be reduced to $O(p^2)$ using randomization.*

5.2. Abelian Runs

Let us first slightly modify the algorithm presented in the previous section. Observe that whenever we start a new phase having just read a block $w[i..i+p-1]$, instead of performing the computations using a simple procedure described above, we could launch the algorithm of Section 4.1 for $w[\max(i-p, 0)..]$ and $\mathcal{P} = \mathcal{P}_{w[i..i+p-1]}$, simulate it until it needs to read $w[i+p]$ and then feed it with newly read letters until the maximal extension of $w[i..i+p-1]$ anchored at k is found (i.e., until the respective entry of the B array is set to ∞). Other anchored runs output by the algorithm should be ignored, of course. As before, if such a process is running while we have completed reading a subsequent block, we do not start a new phase.

It is easy to see that such an algorithm is equivalent to the previous one. However, if we use the algorithm of Section 4.2 instead, we automatically get a possibility to check whether the maximal extension of $w[i..i+p-1]$ anchored at k is a maximal fragment with period $\mathcal{P}_{w[i..i+p-1]}$. Note that we start the simulation at a position $\max(i-p, 0)$ which is smaller than b_{i-1}, unless the latter is 0. This guarantees that left-maximality is correctly verified despite the fact the fragment prior to position $i-p$ is ignored in the simulation. As before, we disregard any other abelian run that the algorithm of Section 4.2 may return. We run this process in parallel for all possible anchors to guarantee that each abelian run with period of norm p is reported exactly once. More precisely, in ambiguous cases a run is reported for the anchor corresponding to the factorization with shortest tail, just as in Section 4.2.

Theorem 15. *There is an algorithm which computes online all the abelian runs with periods of norm p in a word w of length n over an alphabet of size σ in time $O(np)$ and additional space $O(p(\sigma+p))$, which can be reduced to $O(p^2)$ using randomization.*

6. Offline Algorithm for Computing All Abelian Runs

In this section we present an $O(n^2)$-time offline algorithm which computes all the abelian runs. As a starting point, we use the set of all anchored runs computed by the algorithm by Matsuda et al. (see Section 3). Recall that all abelian runs are anchored runs with the same period. Hence, it suffices to filter out those anchored runs which are properly contained in another anchored run with the same period. We also need to make sure that every abelian run is reported once only (despite possibly being k-anchored for different anchors k).

Note that this filtering can be performed independently for distinct periods. If we have a list of anchored runs with a fixed period, sorted by the starting position, it is easy to retrieve the abelian runs of that period with a single scan of the list. Ordering by the starting position can be performed together for all periods so that it takes $O(r+n)$ time where r is the number of all anchored runs. Hence, the main difficulty is grouping according to the period. For this, we shall assign to each fragment of w an *identifier*, so that two fragments are abelian-equivalent if and only if their identifiers are equal. The identifiers of periods can be easily retrieved since given a k-anchored run, we can easily locate one of the cores of the underlying factorization.

Thus, in the remaining part of this section we design a naming algorithm which assigns the identifiers. A naive solution would be to generate the Parikh vectors of all substrings of w, sort these vectors removing duplicates, and give each fragment a rank of its Parikh vector in that order. However, already storing the Parikh vectors can take prohibitive $\Theta(n^2\sigma)$ space.

To overcome this issue, we use the concept of *diff-representation*, originally introduced in the context of abelian periods [10]. Observe that in a sense the Parikh vectors of fragments can be generated efficiently: for a fixed p, we can first generate $\mathcal{P}_{w[0..p-1]}$, then update it to $\mathcal{P}_{w[1..p]}$, and so on until we reach $\mathcal{P}_{w[n-p..n-1]}$. In other words, the Parikh vectors of all fragments of length p can be represented in a sequence so that the total Hamming distance of the adjacent vectors is $O(n)$. The diff-representation, designed to manipulate sequences satisfying such a property, is formally defined as a sequence of single-entry changes such that the original sequence of vectors is a subsequence of intermediate results when applying this operations starting from the null vector (of the fixed dimension r). Note that the diff-representation of a sequence of Parikh vectors of all fragments of w can be computed in time $O(n^2)$ proportional to its size. The following result lets us efficiently assign identifiers to its elements.

Lemma 16 ([10]). *Given a sequence of vectors of dimension r represented using a diff-representation of size m, consider the problem of assigning integer identifiers of size $n^{O(1)}$ so that equality of vectors is equivalent to equality of their identifiers. It can be solved in $O(r+m\log r)$ time using a deterministic algorithm and in $O(r+m)$ time using a Monte Carlo algorithm which is correct with high probability ($1-\frac{1}{(r+m)^c}$ where c can be chosen arbitrarily large).*

In our setting this yields the following result.

Theorem 17. *There exists an $O(n^2)$-time randomized algorithm (Monte Carlo, correct with high-probability) which computes all abelian runs in a given word of length n. Additionally, there exists an $O(n^2 \log \sigma)$-time deterministic algorithm solving the same problem.*

7. Conclusions

We gave algorithms that, given a word w of length n over an alphabet of cardinality σ, return all the abelian runs of a given period $\mathcal{P}$ in w in time $O(n)$ and space $O(\sigma + p)$, or all the abelian runs with periods of a given norm p in time $O(np)$ and space $O(p(\sigma + p))$. These algorithms work in an online manner. We also presented an $O(n^2)$ (resp. $O(n^2 \log n)$)-time offline randomized (resp. deterministic) algorithm for computing all the abelian runs in a word of length n. One may wonder if it is possible to reduce further the complexities of these latter algorithms. We believe that further combinatorial results on the structure of the abelian runs in a word could lead to novel solutions.

References

[1] W. F. Smyth, Computing regularities in strings: a survey, European Journal of Combinatorics 34 (1) (2013) 3–14. doi:10.1016/j.ejc.2012.07.010.

[2] R. Kolpakov, G. Kucherov, Finding maximal repetitions in a word in linear time, in: 40th Annual Symposium on Foundations of Computer Science, FOCS 1999, IEEE Computer Society, New-York, 1999, pp. 596–604. doi:10.1109/SFFCS.1999.814634.

[3] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, K. Tsuruta, A new characterization of maximal repetitions by Lyndon trees, in: P. Indyk (Ed.), Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015, SIAM, 2015, pp. 562–571. doi:10.1137/1.9781611973730.38.

[4] H. Bannai, T. I, S. Inenaga, Y. Nakashima, M. Takeda, K. Tsuruta, The "runs" theorem, CoRR abs/1406.0263v7. URL http://arxiv.org/abs/1406.0263v7

[5] S. Constantinescu, L. Ilie, Fine and Wilf's theorem for abelian periods, Bulletin of the European Association for Theoretical Computer Science 89 (2006) 167–170.

[6] S. Matsuda, S. Inenaga, H. Bannai, M. Takeda, Computing abelian covers and abelian runs, in: J. Holub, J. Zdárek (Eds.), Prague Stringology Conference, PSC 2014, Czech Technical University in Prague, 2014, pp. 43–51.

[7] G. Fici, T. Lecroq, A. Lefebvre, É. Prieur-Gaston, Online computation of abelian runs, in: A. H. Dediu, E. Formenti, C. Martín-Vide, B. Truthe (Eds.), Language and Automata Theory and Applications, LATA 2015, Vol. 8977 of LNCS, Springer International Publishing, 2015, pp. 391–401. doi:10.1007/978-3-319-15579-1_30.

[8] L. J. Cummings, W. F. Smyth, Weak repetitions in strings, Journal of Combinatorial Mathematics and Combinatorial Computing 24 (1997) 33–48.

[9] M. Dietzfelbinger, A. R. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, R. E. Tarjan, Dynamic perfect hashing: Upper and lower bounds, SIAM Journal on Computing 23 (4) (1994) 738–761. doi:10.1137/S0097539791194094.

[10] T. Kociumaka, J. Radoszewski, W. Rytter, Fast algorithms for abelian periods in words and greatest common divisor queries, in: N. Portier, T. Wilke (Eds.), 30th International Symposium on Theoretical Aspects of Computer Science, STACS 2013, Vol. 20 of LIPIcs, Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2013, pp. 245–256. doi:10.4230/LIPIcs.STACS.2013.245.